17 | Python编译器（一）：如何用工具生成编译器？

2020-07-10 宫文学

编译原理实战课 进入课程 >

讲述：宫文学
时长 22:17 大小 20.42M

你好，我是宫文学。

最近几年，Python 在中国变得越来越流行，我想可能有几个推动力：第一个是因为人工智能热的兴起，用 Python 可以很方便地使用流行的 AI 框架，比如 TensorFlow；第二个重要的因素是编程教育，特别是很多面对青少年的编程课程，都是采用的 Python 语言。

不过，Python 之所以变得如此受欢迎，虽然有外在的机遇，但也得益于它内在的一些优点。比如说：

> Python 的语法比较简单，容易掌握，它强调一件事情只能用一种方法去做。对于老一代的程序员来说，Python 就像久远的 BASIC 语言，很适合作为初学者的第一门计算机语言去学习，去打开计算机编程这个充满魅力的世界。

Python 具备丰富的现代语言特性，实现方式又比较简洁。比如，它既支持面向对象特性，也支持函数式编程特性，等等。这对于学习编程很有好处，能够带给初学者比较准确的编程概念。

我个人比较欣赏 Python 的一个原因，是它能够充分利用开源世界的一些二进制的库，比如说，如果你想研究计算机视觉和多媒体，可以用它调用 OpenCV 和 FFmpeg。Python 跟 AI 框架的整合也是同样的道理，这也是 Python 经常用于系统运维领域的原因，因为它很容易调用操作系统的一些库。

最后，Python 还有便于扩展的优势。如果你觉得 Python 有哪方面能力的不足，你也可以用 C 语言来写一些扩展。而且，你不仅仅可以扩展出几个函数，你还能扩展出新的类型，并在 Python 里使用这些新类型。比如，Python 的数学计算库是 NumPy，它的核心代码是用 C 语言编写的，性能很高。

看到这里，你自然会好奇，这么一门简洁有力的语言，是如何实现的呢？吉多·范罗苏姆（Python 初始设计者）在编写 Python 的编译器的时候，脑子里是怎么想的呢？

从这一讲开始，我们就进入到 Python 语言的编译器内部，去看看它作为一门动态、解释执行语言的代表，是如何做词法分析、语法分析和语义分析的，又是如何解释执行的，以及它的运行时有什么设计特点，让它可以具备这些优势。你在这个过程中，也会对编译技术的应用场景了解得更加全面。这也正是我要花 3 讲的时间，带领你来解析 Python 编译器的主要原因。

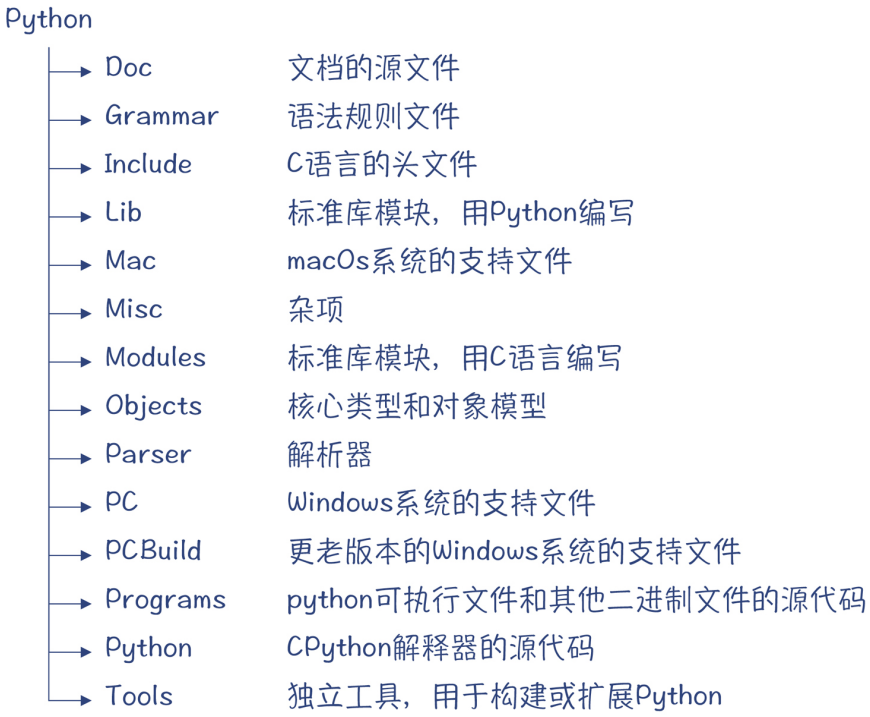
今天这一讲，我们重点来研究 Python 的词法分析和语法分析功能，一起来看看它在这两个处理阶段都有什么特点。你会学到一种新的语法分析实现思路，还能够学到 CST 跟 AST 的区别。

好了，让我们开始吧。

编译源代码，并跟踪调试

首先，你可以从 python.org 网站下载 3.8.1 版本的源代码。解压后你可以先自己浏览一下，看看能不能找到它的词法分析器、语法分析器、符号表处理程序、解释器等功能的代码。

Python 源代码划分了多个子目录，每个子目录的内容整理如下：



首先，你会发现 Python 编译器是用 C 语言编写的。这跟 Java、Go 的编译器不同，Java 和 Go 语言的编译器是支持自举的编译器，也就是这两门语言的编译器是用这两门语言自身实现的。

实际上，用 C 语言实现的 Python 编译器叫做 **CPython**，是 Python 的几个编译器之一。它的标准库也是由 C 语言和 Python 混合编写的。**我们课程中所讨论的就是 CPython，它是 Python 语言的参考实现，也是 macOS 和 Linux 缺省安装的版本。**

不过，Python 也有一个编译器是用 Python 本身编写的，这个编译器是 PyPy。它的图标是一条咬着自己尾巴的衔尾蛇，表明这个编译器是自举的。除此之外，还有基于 JVM 的 Jython，这个版本的优势是能够借助成熟的 JVM 生态，比如可以不用自己写垃圾收集器，还能够调用丰富的 Java 类库。如果你觉得理解 C 语言的代码比较困难，你也可以去看看这两个版本的实现。

在 Python 的“开发者指南”网站上，有不少关于 Python 内部实现机制的技术资料。**请注意**，这里的开发者，指的是有兴趣参与 Python 语言开发的程序员，而不是 Python 语言的使用者。这就是像 Python 这种开源项目的优点，它欢迎热爱 Python 的程序员来修改和增强 Python 语言，甚至你还可以增加一些自己喜欢的语言特性。

根据开发者指南的指引，你可以编译一下 Python 的源代码。注意，你要用**调试模式**来编译，因为接下来我们要跟踪 Python 编译器的运行过程。这就要使用**调试工具 GDB**。

GDB 是 GNU 的调试工具，做 C 语言开发的人一般都会使用这个工具。它支持通过命令行调试程序，包括设置断点、单步跟踪、观察变量的值等，这跟你在 IDE 里调试程序的操作很相似。

开发者指南中有如何用调试模式编译 Python，并如何跟 GDB 配合使用的信息。实际上，GDB 现在可以用 Python 来编写扩展，从而给我们带来更多的便利。比如，我们在调试 Python 编译器的时候，遇到 Python 对象的指针（PyObject*），就可以用更友好的方式来显示 Python 对象的信息。

好了，接下来我们就通过跟踪 Python 编译器执行过程，看看它在编译过程中都涉及了哪些主要的程序模块。

在 tokenizer.c 的 tok_get() 函数中打一个断点，通过 GDB 观察 Python 的运行，你会发现下面的调用顺序（用 bt 命令打印输出后整理的结果）：

程序文件	函数	说明
Pragrams/python.c	main()	Python解释器的入口
Modules/main.c	Py_BytesMain()	
Modules/main.c	pymain_main()	
Modules/main.c	Py_RunMain()	
Modules/main.c	pymain_run_python()	编译执行代码，或者启动REPL
Modules/main.c	pymain_run_file()	从文件读入代码，并执行
Python/pythonrun.c	PyRun_AnyFileExFlags()	执行Python文件
Python/pythonrun.c	PyRun_SimpleFileExFlags()	
Python/pythonrun.c	PyRun_FileExFlags()	
Parser/parsetok.c	PyParser_ASTFromFileObject()	生成AST
Parser/parsetok.c	PyParser_ParseFileObject()	生成解析树
Parser/parsetok.c	parsetok()	获取Token串，并解析
Parser/tokenizer.c	PyTokenizer_Get()	词法分析
Parser/tokenizer.c	tok_get()	词法分析

这个过程是运行 Python 并执行到词法分析环节，你可以看到完整的程序执行路径：

1. 首先是 python.c，这个文件很短，只是提供了一个 main() 函数。你运行 python 命令的时候，就会先进入这里。

2. 接着进入 Modules/main.c 文件，这个文件里提供了运行环境的初始化等功能，它能执行一个 python 文件，也能启动 REPL 提供一个交互式界面。
3. 之后是 Python/pythonrun.c 文件，这是 Python 的解释器，它调用词法分析器、语法分析器和字节码生成功能，最后解释执行。
4. 再之后来到 Parser 目录的 parsetok.c 文件，这个文件会调度词法分析器和语法分析器，完成语法分析过程，最后生成 AST。
5. 最后是 toknizer.c，它是词法分析器的具体实现。

拓展：REPL 是 Read-Evaluate-Print-Loop 的缩写，也就是通过一个交互界面接受输入并回显结果。

通过上述的跟踪过程，我们就进入了 Python 的词法分析功能。下面我们就来看一下它是怎么实现的，再一次对词法分析的原理做一下印证。

Python 的词法分析功能

首先，你可以看一下 tokenizer.c 的 tok_get() 函数。你一阅读源代码，就会发现，这是我们很熟悉的一个结构，它也是通过有限自动机把字符串变成 Token。

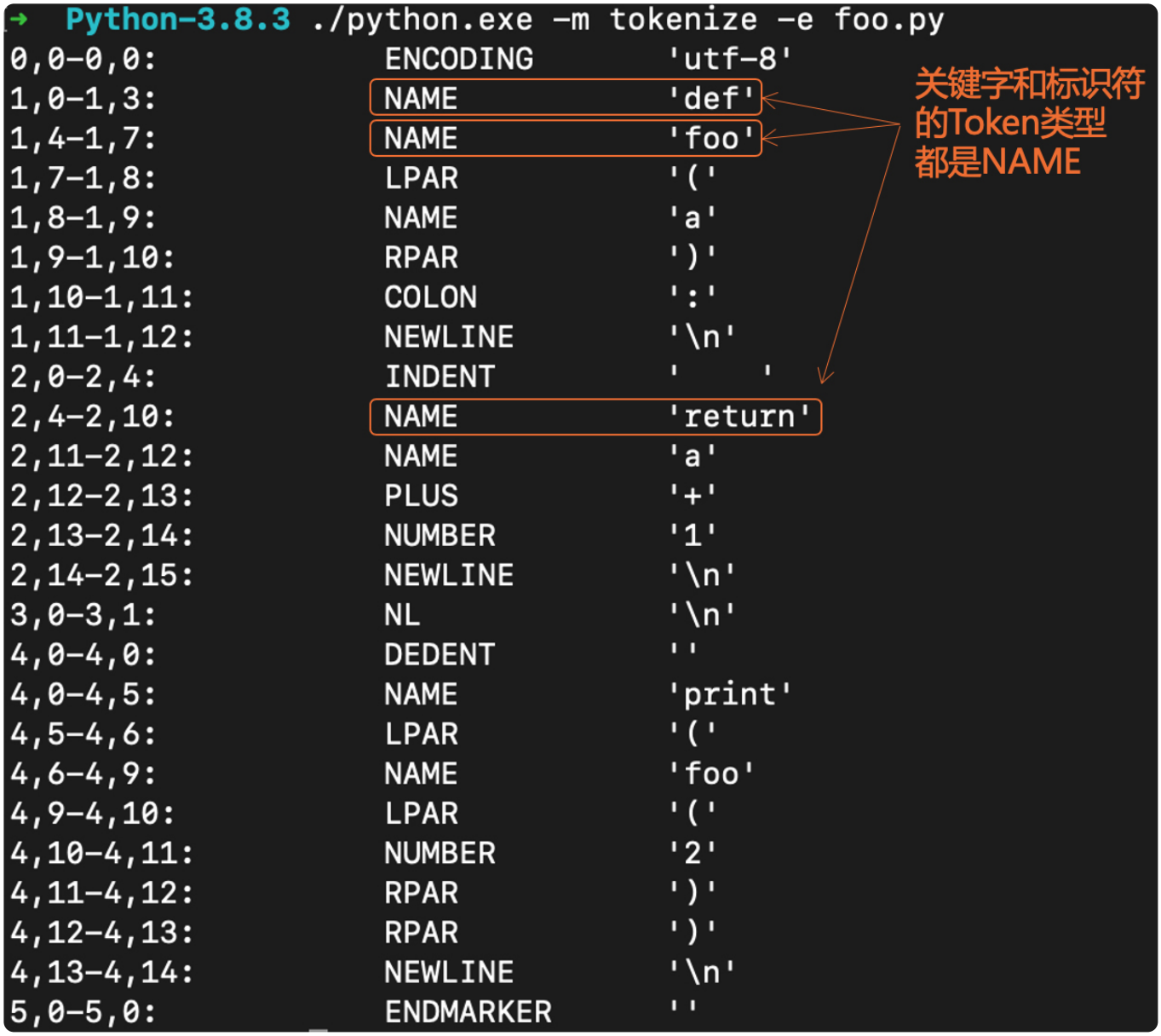
你还可以用另一种更直接的方法来查看 Python 词法分析的结果。

复制代码

```
./python.exe -m tokenize -e foo.py
```

补充：其中的 python.exe 指的是 Python 的可执行文件，如果是在 Linux 系统，可执行文件是 python。

运行上面的命令会输出所解析出的 Token：



其中的第二列是 Token 的类型，第三列是 Token 对应的字符串。各种 Token 类型的定义，你可以在 Grammar/Tokens 文件中找到。

我们曾在研究Java 编译器的时候，探讨过如何解决关键字和标识符的词法规则冲突的问题。**那么 Python 是怎么实现的呢？**

原来，Python 在词法分析阶段根本没有区分这两者，只是都是作为“NAME”类型的 Token 来对待。

补充：Python 里面有两个词法分析器，一个是用 C 语言实现的（tokenizer.c），一个是用 Python 实现的（tokenizer.py）。C 语言版本的词法分析器由编译器使用，性能更高。

所以，Python 的词法分析功能也比较常规。其实你会发现，每个编译器的词法分析功能都大同小异，你完全可以借鉴一个比较成熟的实现。Python 跟 Java 的编译器稍微不同的一点，就是没有区分关键字和标识符。

接下来，我们来关注下这节课的重点内容：语法分析功能。

Python 的语法分析功能

在 GDB 中继续跟踪执行过程，你会在 parser.c 中找到语法分析的相关逻辑：

...	...	
Parser/parsetok.c	parsetok()	获取Token串，并解析
Parser/parser.c	PyParser_AddToken()	语法分析

那么，Python 的语法分析有什么特点呢？它采用的是什么算法呢？是自顶向下的算法，还是自底向上的算法？

首先，我们到 Grammar 目录，去看一下 Grammar 文件。这是一个用 EBNF 语法编写的 Python 语法规则文件，下面是从中节选的几句，你看是不是很容易读懂呢？

复制代码

```
//声明函数
funcdef: 'def' NAME parameters ['->' test] ':' [TYPE_COMMENT] func_body_suite
//语句
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | del_stmt | pass_stmt | flow_stmt |
             import_stmt | global_stmt | nonlocal_stmt | assert_stmt)
```

通过阅读规则文件，你可以精确地了解 Python 的语法规则。

这个规则文件是给谁用的呢？实际上 Python 的编译器本身并不使用它，它是给一个pgen 的工具程序（Parser/pgen）使用的。这个程序能够基于语法规则生成**解析表**（Parse Table），供语法分析程序使用。有很多工具能帮助你生成语法解析器，包括 yacc（GNU 版本是 bison）、ANTLR 等。

有了 pgen 这个工具，你就可以通过修改规则文件来修改 Python 语言的语法，比如，你可以把函数声明中的关键字“def”换成“function”，这样你就可以用新的语法来声明函数。

pgen 能给你生成新的语法解析器。parser.c 的注释中讲解了它的工作原理。它是把 EBNF 转化成一个 NFA，然后再把这个 NFA 转换成 DFA。基于这个 DFA，在读取 Token 的时候，编译器就知道如何做状态迁移，并生成解析树。

这个过程你听上去是不是有点熟悉？实际上，我们在第 2 讲讨论正则表达式工具的时候，就曾经把正则表达式转化成了 NFA 和 DFA。基于这个技术，我们既可以做词法解析，也可以做语法解析。

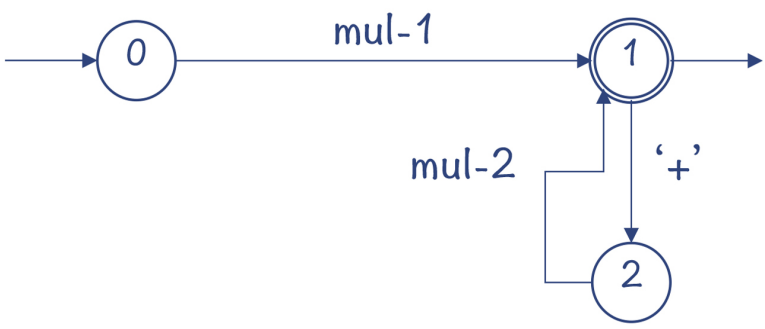
实际上，Python 用的是 LL(1) 算法。我们来回忆一下 LL(1) 算法的特点：**针对每条语法规则，最多预读一个 Token，编译器就可以知道该选择哪个产生式。**这其实就是一个 DFA，从一条语法规则，根据读入的 Token，迁移到下一条语法规则。

我们通过一个例子来看一下 Python 的语法分析特点，这里采用的是我们熟悉的一个语法规则：

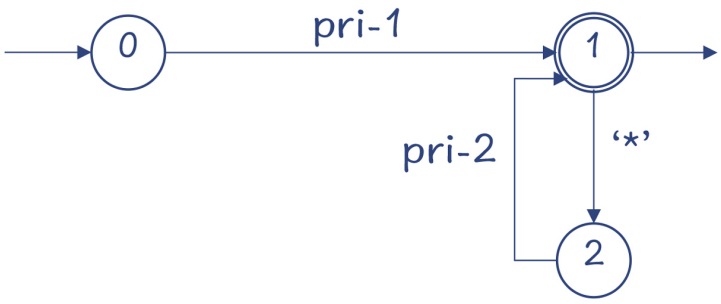
复制代码

```
1 add: mul ('+' mul)*
2 mul: pri ('*' pri)*
3 pri: IntLiteral | '(' add ')'
```

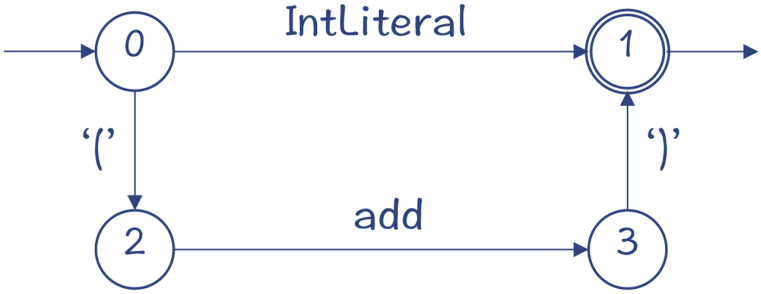
我把这些语法规则对应的 DFA 画了出来。你会看到，它跟采用递归下降算法的思路是一样的，只不过换了种表达方式。



add: mul ('+' mul)*对应的DFA



mul: pri ('*' pri)*对应的DFA



pri: IntLiteral | '(' add ')'对应的DFA

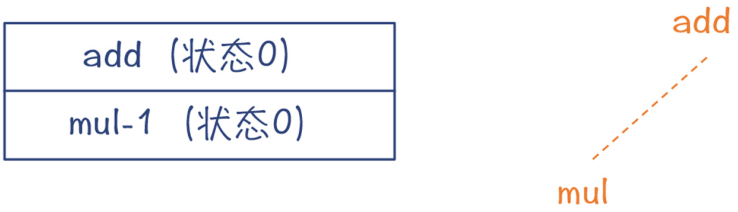
不过，跟手写的递归下降算法为解析每个语法规则写一个函数不同，parser.c 用了一个通用的函数去解析所有的语法规则，它所依据的就是为每个规则所生成的 DFA。

主要的实现逻辑是在 parser.c 的 PyParser_AddToken() 函数里，你可以跟踪它的实现过程。为了便于你理解，我模仿 Python 编译器，用上面的文法规则解析了一下“2+3*4+5”，并把整个解析过程画成图。

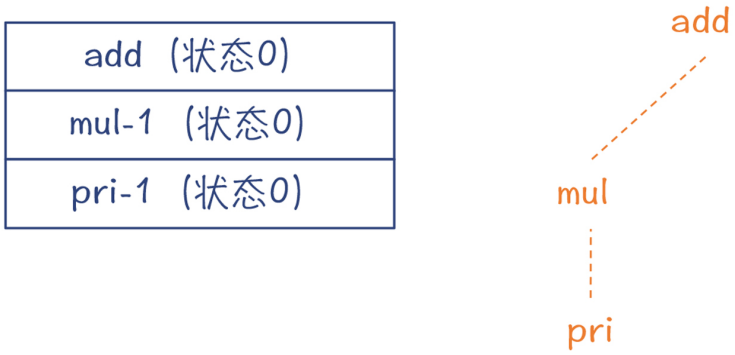
在解析的过程，我用了一个栈作为一个工作区，来保存当前解析过程中使用的 DFA。

第 1 步，匹配 add 规则。把 add 对应的 DFA 压到栈里，此时该 DFA 处于状态 0。这时候预读了一个 Token，是字面量 2。

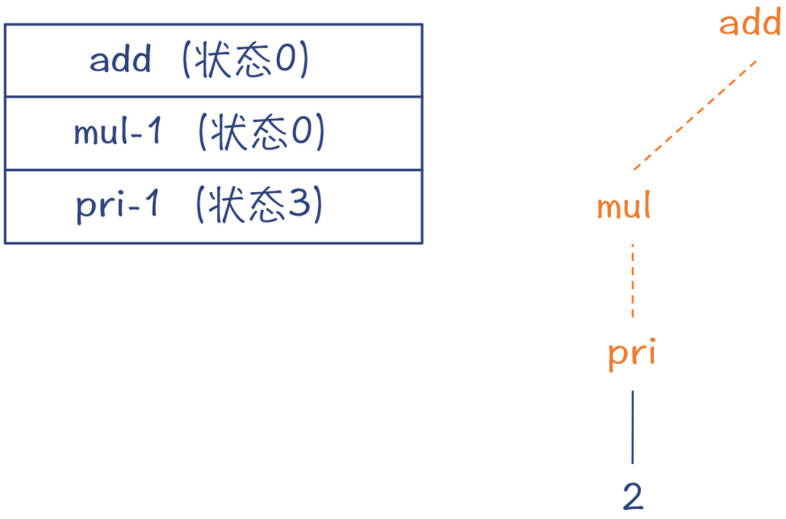
第 2 步，根据 add 的 DFA，走 mul-1 这条边，去匹配 mul 规则。这时把 mul 对应的 DFA 入栈。在示意图中，栈是从上往下延伸的。



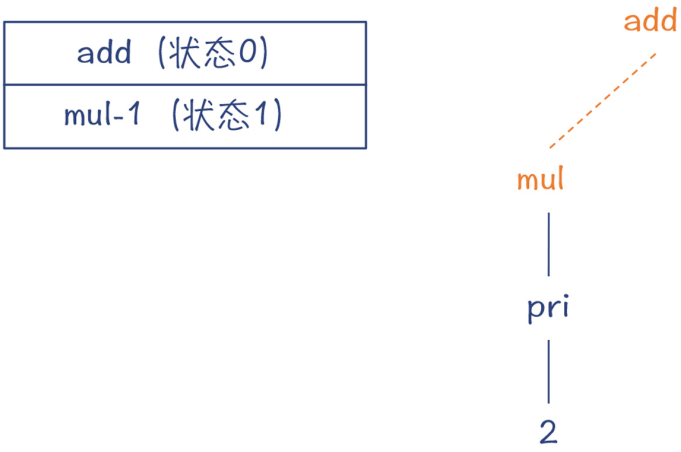
第 3 步，根据 mul 的 DFA，走 pri-1 这条边，去匹配 pri 规则。这时把 pri 对应的 DFA 入栈。



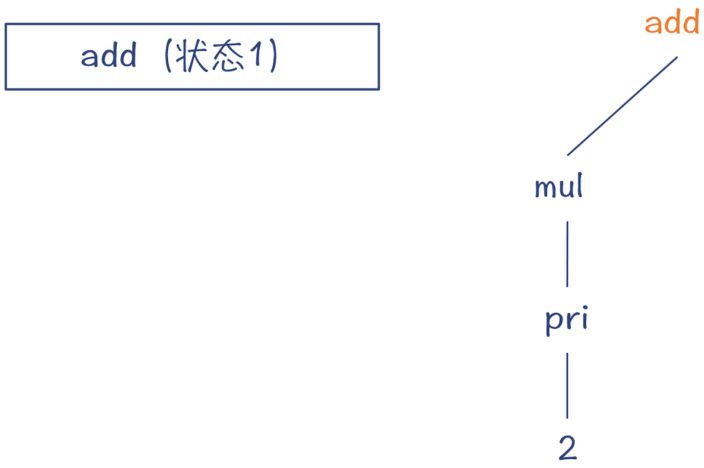
第 4 步，根据 pri 的 DFA，因为预读的 Token 是字面量 2，所以移进这个字面量，并迁移到状态 3。同时，为字面量 2 建立解析树的节点。这个时候，又会预读下一个 Token，'+'号。



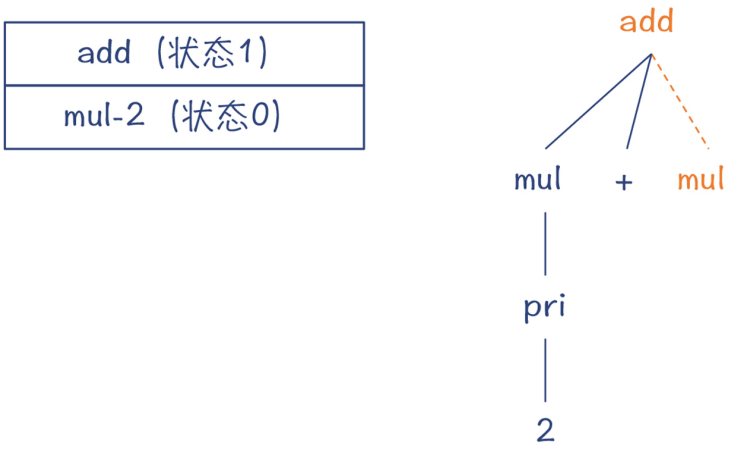
第 5 步，从栈里弹出 pri 的 DFA，并建立 pri 节点。因为成功匹配了一个 pri，所以 mul 的 DFA 迁移到状态 1。



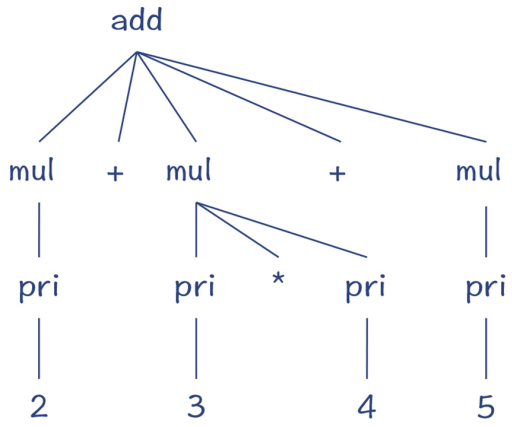
第 6 步，因为目前预读的 Token 是'+'号，所以 mul 规则匹配完毕，把它的 DFA 也从栈里弹出。而 add 对应的 DFA 也迁移到了状态 1。



第 7 步，移进'+'号，把 add 的 DFA 迁移到状态 2，预读了下一个 Token：字面量 3。这个 Token 是在 mul 的 First 集合中的，所以就走 mul-2 边，去匹配一个 mul。



按照这个思路继续做解析，直到最后，可以得到完整的解析树：



总结起来，Python 编译器采用了一个通用的语法分析程序，以一个栈作为辅助的数据结构，来完成各个语法规则的解析工作。当前正在解析的语法规则对应的 DFA，位于栈顶。一旦当前的语法规则匹配完毕，那语法分析程序就可以把这个 DFA 弹出，退回到上一级的语法规则。

所以说，语法解析器生成工具，会基于不同的语法规则来生成不同的 DFA，但语法解析程序是不变的。这样，你随意修改语法规则，都能够成功解析。

上面我直观地给你解读了一下解析过程。你可以用 GDB 来跟踪一下 PyParser_AddToken() 函数，从而了解得更具体。你在这个函数里，还能够看到像下面这样的语句，这是对外输出调试信息。

复制代码

```
1 D(printf(" Push '%s'\n", d1->d_name));    //把某DFA入栈
```

你还可以用“-d”参数运行 python，然后在 REPL 里输入程序，这样它就能打印出这些调试信息，包括什么时候把 DFA 入栈、什么时候出栈，等等。我截取了一部分输出信息，你可以看一下。

在 Python 的语法规则里，arith_expr 指的是加减法的表达式，term 指的是乘除法的表达式，atom 指的是基础表达式。这套词汇也经常被用于语法规则中，你可以熟悉起来。

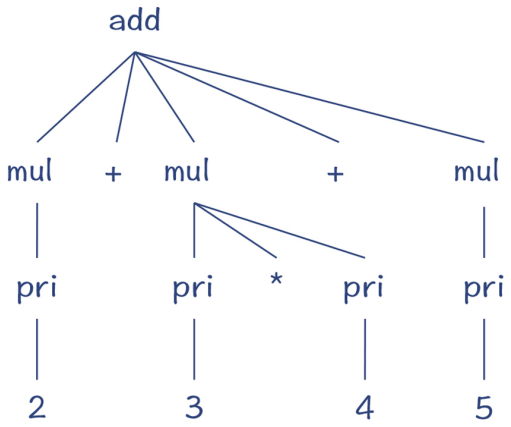
好了，现在你已经知道了语法解析的过程。不过你可能注意到了，上面的语法解析过程形成的结果，我没有叫做是 AST，而是叫做**解析树**（Parse Tree）。看到这里，你可能会产生疑问：**解析源代码不就会产生 AST 吗？怎么这里是生成一个叫做解析树的东西？什么是解析树，它跟 AST 有啥区别？**别着急，下面我就来为你揭晓答案。

解析树和 AST 的区别

解析树又可以叫做 **CST**（Concret Syntax Tree，具体语法树），与 AST（抽象语法树）是相对的：一个具体，一个抽象。

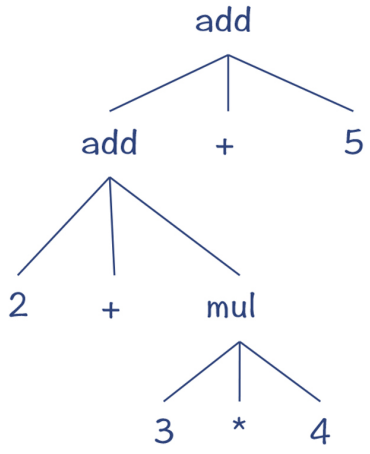
它俩的区别在于：**CST 精确地反映了语法规则的推导过程，而 AST 则更准确地表达了程序的结构。如果说 CST 是“形似”，那么 AST 就是“神似”。**

你可以看看在前面的这个例子中，所形成的 CST 的特点。



首先，加法是个二元运算符，但在这里 add 节点下面对应了两个加法运算符，跟原来加法的语义不符。第二，很多节点都只有一个父节点，这个其实可以省略，让树结构更简洁。

所以，我们期待的 AST 其实是这样的：



这就是 CST 和 AST 的区别。

理解了这个知识点以后，我们拿 Python 实际的 CST 和 AST 来做一下对比。在 Python 的命令行中，输入下面的命令：

复制代码

```
>>> from pprint import pprint
>>> import parser
>>> cst = parser.expr('2+3+4')  //对加法表达式做解析
>>> pprint(parser.st2list(cst)) //以美观的方式打印输出CST
```

你会得到这样的输出结果：

```
>>> from pprint import pprint
>>> import parser
>>> cst = parser.expr('2+3*4+5')
>>> pprint(parser.st2list(cst))
[258,
 [332,
  [306,
   [310,
    [311,
     [312,
      [313,
       [316,
        [317,
         [318,
          [319,
           [320,
            [321, [322, [323, [324, [325, [2, '2']]]]]],
            [14, '+'],
            [321,
             [322, [323, [324, [325, [2, '3']]]]],
             [16, '*'],
             [322, [323, [324, [325, [2, '4']]]]],
            [14, '+'],
            [321, [322, [323, [324, [325, [2, '5']]]]]]]]]]]]]]]],
 [4, ''],
 [0, '']]
```

这是用缩进的方式显示了 CST 的树状结构，其中的数字是符号和 Token 的编号。你可以从 Token 的字典（dict）里把它查出来，从而以更加直观的方式显示 CST。

我们借助一个 lex 函数来做美化的工作。现在再显示一下 CST，就更直观了：

```
>>> pprint(lex('2+3*4+5'))
['eval_input',
 ['testlist',
  ['test',
   ['or_test',
    ['and_test',
     ['not_test',
      ['comparison',
       ['expr',
        ['xor_expr',
         ['and_expr',
          ['shift_expr',
           ['arith_expr',
            ['term',
             ['factor', ['power', ['atom_expr', ['atom', ['NUMBER', '2']]]]]],
            ['PLUS', '+'],
            ['term',
             ['factor', ['power', ['atom_expr', ['atom', ['NUMBER', '3']]]]],
             ['STAR', '*'],
             ['factor', ['power', ['atom_expr', ['atom', ['NUMBER', '4']]]]],
            ['PLUS', '+'],
            ['term',
             ['factor',
              ['power', ['atom_expr', ['atom', ['NUMBER', '5']]]]]]]]]]]]]]]]],
 ['NEWLINE', ''],
 ['ENDMARKER', '']]
```

那么，Python 把 CST 转换成 AST，会是什么样子呢？

你可以在命令行敲入下面的代码，来显示 AST。它虽然是以文本格式显示的，但你能发现它是一个树状结构。这个树状结构就很简洁：

```
>>> import ast
>>> t=ast.parse('2+3*4+5')
>>> print(ast.dump(t))
Module(body=[Expr(value=BinOp(left=BinOp(left=Constant(value=2, kind=None), op
=Add(), right=BinOp(left=Constant(value=3, kind=None), op=Mult(), right=Consta
nt(value=4, kind=None))), op=Add(), right=Constant(value=5, kind=None)))], typ
e_ignores=[])
```

如果你嫌这样不够直观，还可以用另一个工具“instaviz”，在命令行窗口用 pip 命令安装 instaviz 模块，以图形化的方式更直观地来显示 AST。instaviz 是“Instant Visualization”（立即可视化）的意思，它能够图形化显示 AST。

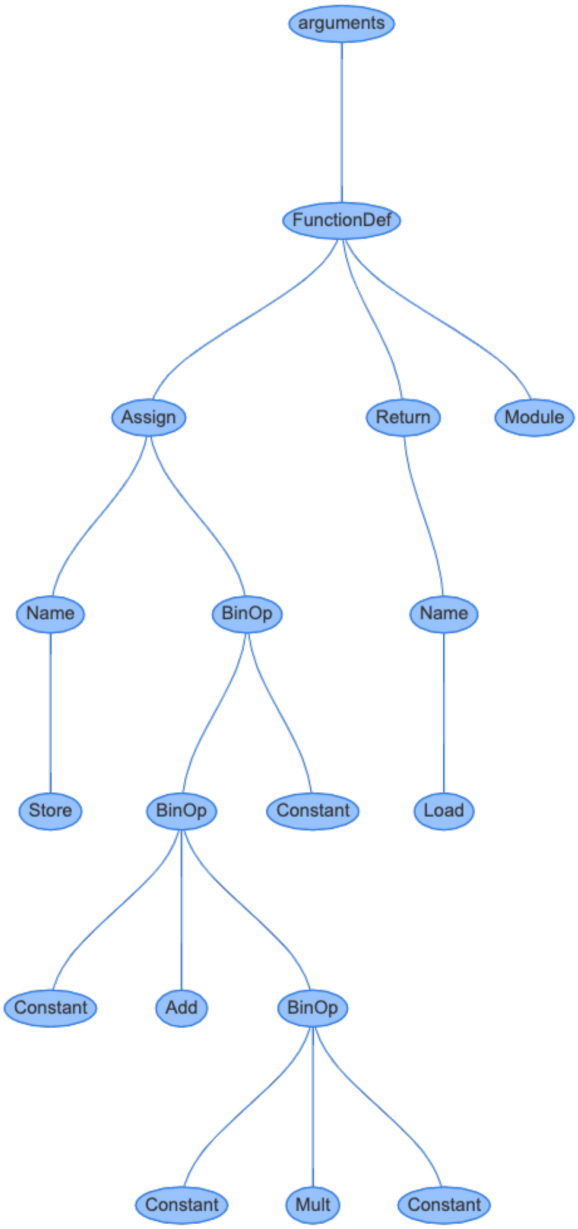
复制代码

```
1 $ pip install instaviz
```

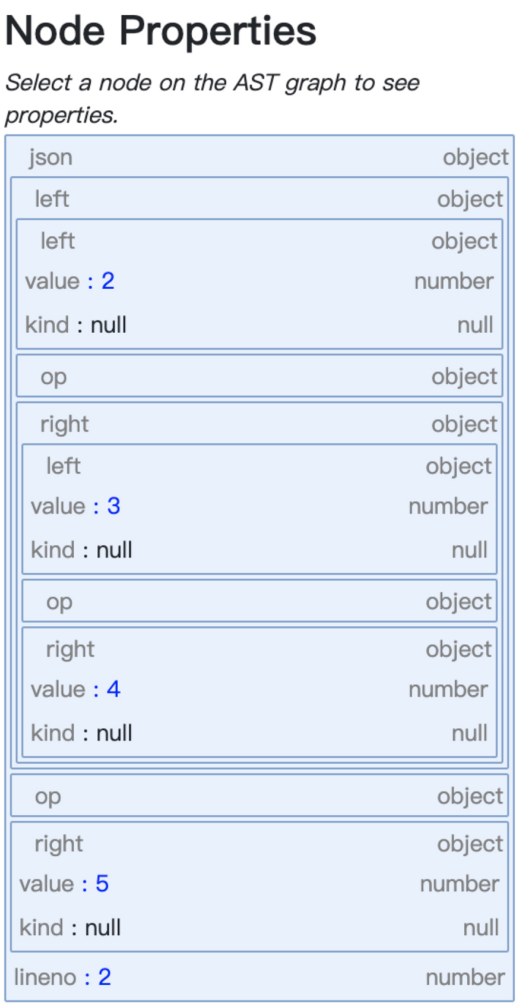
然后启动 Python，并敲入下面的代码：

```
>>> import instaviz
>>> def fun1():
...         a = 2 + 3 * 4 + 5
...         return a
...
>>> instaviz.show(fun1)
Bottle v0.12.18 server starting up (using WSGIRefServer())...
Listening on http://localhost:8080/
Hit Ctrl-C to quit.
```

instaviz 会启动一个 Web 服务器，你可以在浏览器里通过 http://localhost:8080 来访问它，里面有图形化的 AST。你可以看到，这个 AST 比起 CST 来，确实简洁太多了。



点击代表“2+3*4+5”表达式的节点，你可以看到这棵子树的各个节点的属性信息：



总结起来，在编译器里，我们经常需要把源代码转变成 CST，然后再转换成 AST。生成 CST 是为了方便编译器的解析过程。而转换成 AST 后，会让树结构更加精简，并且在语义上更符合语言原本的定义。

那么，Python 是如何把 CST 转换成 AST 的呢？这个过程分为两步。

首先，Python 采用了一种叫做 ASDL 的语言，来定义了 AST 的结构。ASDL是“抽象语法定义语言（Abstract Syntax Definition Language）”的缩写，它可以用于描述编译器中的 IR 以及其他树状的数据结构。你可能不熟悉 ASDL，但可能了解 XML 和 JSON 的 Schema，你可以通过 Schema 来定义 XML 和 JSON 的合法的结构。另外还有 DTD、EBNF 等，它们的作用都是差不多的。

这个定义文件是 Parser/Python.asdl。CPython 编译器中包含了两个程序（Parser/asdl.py 和 Parser/asdl_c.py）来解析 ASDL 文件，并生成 AST 的数据结构。最

后的结果在 Include/Python-ast.h 文件中。

到这里，你可能会有疑问：**这个 ASDL 文件及解析程序不就是生成了 AST 的数据结构吗？为什么不手工设计这些数据结构呢？有必要采用一种专门的 DSL 来做这件事情吗？**

确实如此。Java 语言的 AST，只是采用了手工设计的数据结构，也没有专门用一个 DSL 来生成。

但 Python 这样做确实有它的好处。上一讲我们说过，Python 的编译器有多种语言的实现，因此基于统一的 ASDL 文件，我们就可以精准地生成不同语言下的 AST 的数据结构。

在有了 AST 的数据结构以后，**第二步，是把 CST 转换成 AST，这个工作是在 Python/ast.c 中实现的，入口函数是 PyAST_FromNode()。**这个算法是手写的，并没有办法自动生成。

课程小结

今天这一讲，我们开启了对 Python 编译器的探究。我想给你强调下面几个关键要点：

非自举。CPython 的编译器是用 C 语言编写的，而不是用 Python 语言本身。编译器和核心库采用 C 语言会让它性能更高，并且更容易与各种二进制工具集成。

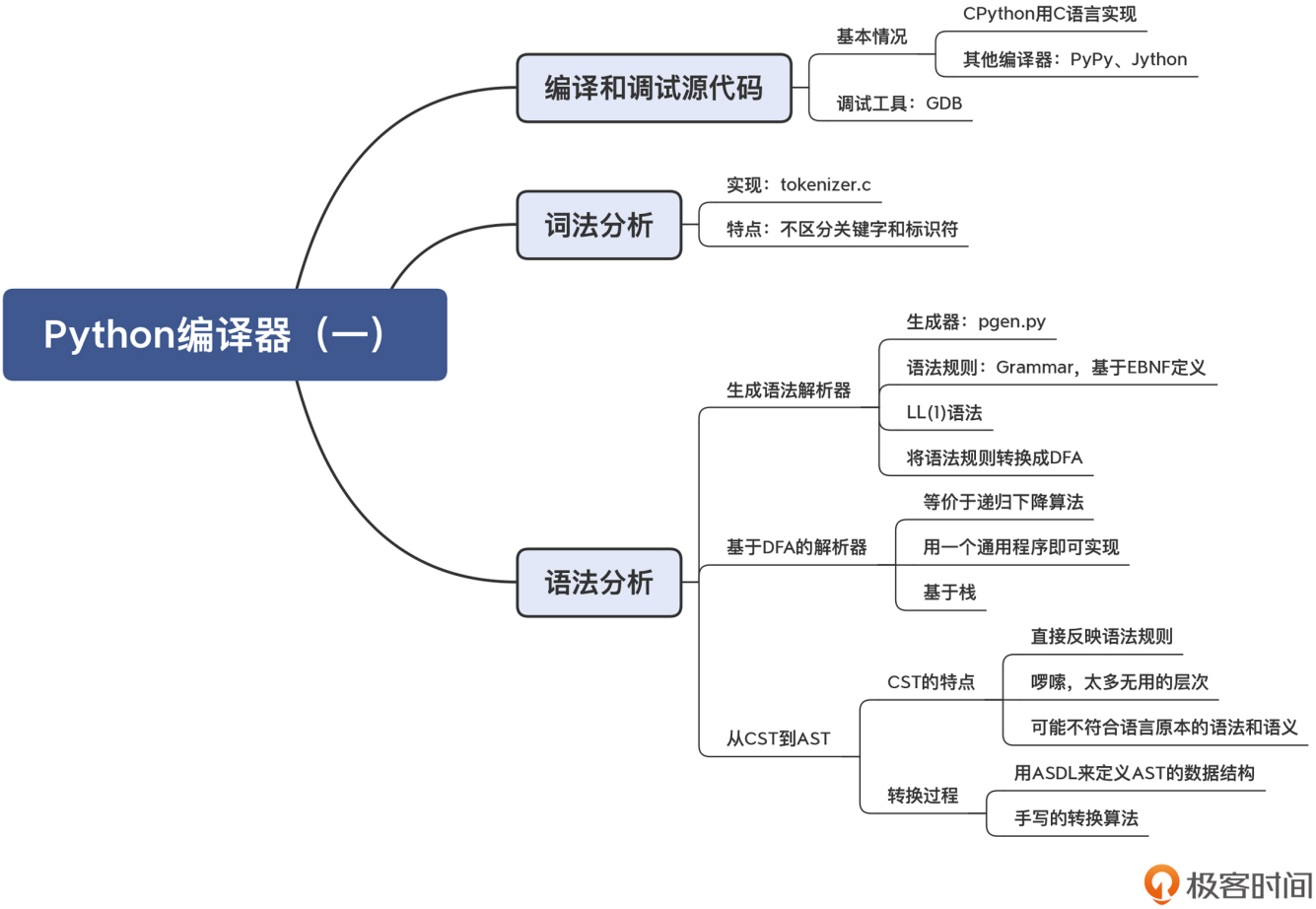
善用 GDB。使用 GDB 可以跟踪 CPython 编译器的执行过程，加深对它的内部机制的理解，加快研究的速度。

编译器生成工具 pgen。pgen 能够根据语法规则生成解析表，让修改语法的过程变得更加容易。

基于 DFA 的语法解析过程。基于 pgen 生成的解析表，通过 DFA 驱动完成语法解析过程，整个执行过程跟递归下降算法的原理相同，但只需要一个通用的解析程序即可。

从 CST 到 AST。语法分析首先生成 CST，接着生成 AST。CST 准确反映了语法推导的过程，但会比较啰嗦，并且可能不符合语义。AST 同样反映了程序的结构，但更简洁，并且支持准确的语义。

本讲的思维导图我也放在这里了，供你参考：



一课一思

这一讲我们提到，Python 的词法分析器没有区分标识符和关键字，但这样为什么没有影响到 Python 的语法分析的功能呢？你可以结合语法规则文件和对语法解析过程的理解，谈谈你的看法。如果你能在源代码里找到确定的答案，那就更好了！

欢迎你在留言区中分享你的见解，也欢迎你把今天的内容分享给更多的朋友，我们下一讲再见。

参考资料

GDB 的安装和配置：参考这篇文章。

提建议

更多课程推荐

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 16 | Java JIT编译器（四）：Graal的后端是如何工作的？

下一篇 18 | Python编译器（二）：从AST到字节码

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。